CSE 220: Systems Fundamentals I
Unit 5: MIPS Assembly: Function Calls and the Stack
Functions

• As you have learned in your programming courses, functions (methods) allow for more structured programs, code reuse, and make code easier to understand and maintain.

• We make a distinction between the **caller** and the **callee**.

• The callee is the function being called.

• The caller is the function that calls the callee.

• In this example, **main** is the caller and **sum** is the callee.

```java
void main() {
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b) {
    return (a + b);
}
```
Functions in MIPS Assembly

- These are the major steps to calling a function in MIPS:
  1. The caller places the parameter values in argument registers – this is where the callee can access them.
  2. The caller saves the return address and transfers control to the callee by jumping to the callee’s label.
  3. The callee allocates (reserves) any memory it may need for local variables.
  4. The callee performs the desired task. The callee must not overwrite registers or memory needed by caller.
  5. The callee places the result in the value register(s), where the calling function can access it (them).
  6. The callee returns control to the caller.
MIPS Function Conventions

- A quick summary/overview before we delve into the details:
  - Call a function: *jump and link* (jal  *label*)
  - Return from a function: *jump register* (jr  $ra)
  - Function arguments: $a0–$a3
  - Return values: $v0,  $v1
  - Local variables: $s0–$s7
  - Temporary values: $t0–$t9
Instruction Support for Functions

• Use jump instructions to call and return from function calls.

• To call a function: `jal function_label` saves PC+4 in register `$ra`, then jumps to the address of the function.
  • It saves PC+4 in `$ra` so that, when the function returns, the next instruction executed is the one immediately following the `jal` instruction

• To return from a function: `jr $ra`
  • Jumps to the address in `$ra`
Input Arguments & Return Value

• C code:

```c
int main() {
    int y;
    ...
    y = diffofsums(2, 3, 4, 5);  // 4 args
    ...
}

int diffofsums(int f, int g, int h, int i) {
    int result;
    result = (f + g) - (h + i);
    return result;             // return value
}
```
Caution!!!

- I am going to show you several attempts at implementing both functions in MIPS
- The first two of attempts will have serious flaws in them, which I will point out
- Attempt #3 will have our final, correct implementation
Functions: Attempt #1 (flawed)

- MIPS assembly code:

```assembly
# $s0 = y
main:

...  
addi $a0, $0, 2    # argument 0 = 2
addi $a1, $0, 3    # argument 1 = 3
addi $a2, $0, 4    # argument 2 = 4
addi $a3, $0, 5    # argument 3 = 5
jal diffofsums     # call Function
add  $s0, $v0, $0  # y = returned value
...```
Functions: Attempt #1 (cont.)

• MIPS assembly code continued...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1 # $t0 = f + g
  add $t1, $a2, $a3 # $t1 = h + i
  sub $s0, $t0, $t1 # result = (f+g)-(h+i)
  add $v0, $s0, $0 # put return value in $v0
  jr $ra # return to caller
Functions: Attempt #1 (cont.)

• MIPS assembly code continued...

# $s0 = result
diffofsums:

    add $t0, $a0, $a1 # $t0 = f + g
    add $t1, $a2, $a3 # $t1 = h + i
    sub $s0, $t0, $t1 # result = (f+g)-(h+i)
    add $v0, $s0, $0   # put return value in $v0
    jr $ra             # return to caller

• diffofsums overwrote 3 registers: $t0, $t1, $s0
• diffofsums can use main memory to temporarily save values of registers, but which ones should/must it save?
The Stack

- The **run-time stack** or **call stack** is a region of memory used to temporarily save variables during function calls.
- A stack is a **last-in-first-out** (LIFO) data structure, meaning that the last item added (**pushed**) onto the stack will be the first one removed (**popped**) from the stack.
- The stack expands by using more memory when more space needed...
- ...and contracts by using less memory when that space is no longer needed.
The Stack

- The **top** of the stack is a memory address and is stored in the `$sp` register ("stack pointer")
- In MIPS, the stack grows **downwards** (from higher to lower memory addresses)
- Left image: before pushing two words onto the stack
- Right image: after pushing the words

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFFC</td>
<td>12345678</td>
<td>7FFFFFFFC</td>
<td>12345678</td>
</tr>
<tr>
<td>7FFFFFF8</td>
<td></td>
<td>7FFFFFF8</td>
<td>AABBCDAD</td>
</tr>
<tr>
<td>7FFFFFF4</td>
<td></td>
<td>7FFFFFF4</td>
<td>11223344</td>
</tr>
<tr>
<td>7FFFFFF0</td>
<td></td>
<td>7FFFFFF0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Kevin McDonnell  Stony Brook University – CSE 220
How Functions Use the Stack

• Called functions must have no unintended side effects
• But `diffofsums` overwrote 3 registers: `$t0, $t1, $s0`

`diffofsums`:

```assembly
add $t0, $a0, $a1  # $t0 = f + g
add $t1, $a2, $a3  # $t1 = h + i
sub $s0, $t0, $t1  # result = (f+g)-(h+i)
add $v0, $s0, $0   # put return value in $v0
jr $ra             # return to caller
```

• These changes might corrupt the `main` function or some other function that calls `diffofsums`
• One way to solve this is to have `diffofsums` save a copy of each register on the stack and restore the values before returning
Functions: Attempt #2 (flawed)

# $s0 = result
diffofsums:
  addi $sp, $sp, -12  # make space on stack
      # to store 3 registers
  sw  $s0, 8($sp)     # save $s0 on stack
  sw  $t0, 4($sp)     # save $t0 on stack
  sw  $t1, 0($sp)     # save $t1 on stack
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  lw  $t1, 0($sp)     # restore $t1 from stack
  lw  $t0, 4($sp)     # restore $t0 from stack
  lw  $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12   # deallocate stack space
  jr  $ra              # return to caller
Functions: Attempt #2 (cont.)

- Left-to-right in the images below:
  a) before the call to `diffofsums`
  b) during the call to `diffofsums`
  c) after the call to `diffofsums`
Functions: Attempt #2 (cont.)

• The stack space that a function allocates *for itself* is called its **stack frame**

• Either the caller or the callee (or both) can create a stack frame
  • Generally, each function should access its own stack frame, but (of course) there is an exception to this that we’ll see a little later

• Now, is a called function supposed to save a copy of every single register it alters?
  • Luckily, the answer is *no*.

• **MIPS register conventions** make a distinction between **preserved** and **non-preserved** registers and data
Register Conventions

<table>
<thead>
<tr>
<th>Preserved</th>
<th>Non-preserved</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Callee-Saved</strong></td>
<td><strong>Caller-Saved</strong></td>
</tr>
<tr>
<td>$s0 -$s7</td>
<td>$t0 -$t9</td>
</tr>
<tr>
<td>$ra</td>
<td>$a0 -$a3</td>
</tr>
<tr>
<td>$sp, $fp</td>
<td>$v0 -$v1</td>
</tr>
<tr>
<td>stack above $sp</td>
<td>stack below $sp</td>
</tr>
</tbody>
</table>

- The $s registers are called **saved registers** because a caller is guaranteed (by convention) that these registers will be saved by the callee.
- The $t registers are called **temporary** because they don’t have to be saved by the callee. If the caller needs to preserve the $t registers, the caller must save them.
Register Conventions

• What about $ra$ and $sp$?

• $sp$ is saved in the sense that the callee deallocates its stack frame before returning by adding the same amount that it subtracted from $sp$ at the beginning of the function.

• The $ra$ register will be overwritten if the callee itself calls a function.

• So if a function “knows” that it is going to call a function, it must save $ra$ along with the $s$ registers before performing the jal.
Nested Function Calls

func1:

```
addi $sp, $sp, -4    # make space on stack
sw       $ra, 0($sp)  # save $ra on stack
jal     func2

...  

lw       $ra, 0($sp)  # restore $s0 from stack
addi $sp, $sp, 4    # deallocate stack space
jr        $ra        # return to caller
```
Functions: Attempt #3 (good)

# $s0 = result
diffofsums:
    addi $sp, $sp, -4  # make space on stack to
        # store one register
    sw  $s0, 0($sp)   # save $s0 on stack
        # no need to save $t0 or $t1
    add  $t0, $a0, $a1  # $t0 = f + g
    add  $t1, $a2, $a3  # $t1 = h + i
    sub  $s0, $t0, $t1  # result = (f+g)-(h+i)
    add  $v0, $s0, $0   # put return value in $v0
    lw   $s0, 0($sp)   # restore $s0 from stack
    addi $sp, $sp, 4   # deallocate stack space
    jr   $ra           # return to caller
Follow the Register Conventions!

• You *must* follow the register conventions when doing your homework

• During grading we will run various automated tests that ensure that your code follows the conventions

• We will also manually inspect your code to ensure that you are following conventions and saving only those registers that should be saved

• For instance, having the callee preserve $t$ registers is bad coding style and does not follow the conventions

• Likewise, having the callee preserve every $s$ register, even those it does not change, is not in accordance with the conventions
Functions Examples

• Let’s look at some example programs that involve function calls
• First, let’s see a program that takes a string and counts the number of vowels in it. We’ll use two functions:
  • count_vowels, which is called by main
  • is_vowel, which is called by count_vowels
• Second, a program that takes a fraction and reduces it to lowest terms
  • reduce, which is called by main
  • gcd, which is called by reduce
• The numerator and denominator will both be divided by their GCD to reduce the fraction to lowest terms
More Than Four Arguments

• What if we want to pass more than four arguments to a function?
  • We can pass the extra arguments on the stack
  • The caller places the extra arguments into its own stack frame
  • The callee will then need to go into the caller’s stack frame to find them
  • This is the one exception to the rule that a function should not access stack data outside its own stack frame
Callee’s Stack Frame

- Figure (a) shows the stack before the callee is invoked
- In figure (b), the callee has been invoked
- The callee saves copies of whatever registers it needs to save and allocates any additional stack space it may need
- Then it will go into the “additional args” section to grab the extra arguments it needs
Example: More Than Four Args.

- Let’s write a program that computes the squared distance between two 3D points \((x_1, y_1, z_1)\) and \((x_2, y_2, z_2)\), which is \(d_{sqr} = (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2\)

- We can pass \(x_1\) through \(a0\), \(y_1\) through \(a1\), \(z_1\) through \(a2\), \(x_2\) through \(a3\), and the values for \(y_2\) and \(z_2\) on the stack.